Semi-Automated Test Generation With SPEST

Adam Mozek
June 2015

Abstract

Automated black box test generation is a useful tool for developers that allows them to quickly generate a large number of tests with a wide range of values tested. The current most popular tool for black box test generation in Java, JML, is lacking in a number of features. SPEST aims to improve upon existing tools, and provide a black box test generation tool that can create human readable tests from simple pre and post conditions added as comments to the Java source code.

Table of Contents

1. Introduction

SPEST is a specification testing language for Java whose goal is to generate black box tests, utilizing techniques from various research journals and proceedings [3, 1, 2] to improve upon existing tools [6]. The language is similar to JMLUnit but makes improvements in the areas of test readability, test generation time, and code littering. SPEST is currently only implemented for the Java programming language, but was initially designed to support generation for multiple languages.

1.1. Description of the Problem

Currently, black box testing generation tools are not as good as they could be. Tools like JML [5] suffer from many issues, among these are the issues of generation of tests taking a long time due to the fact that tests generated by the language follow strict parameters. SPEST aims to alleviate many of the usage problems with JML by providing a more user friendly and simpler to use black box testing language.

JML has the problem of “code littering” wherein annotations must be placed in multiple places throughout the code in order for all dependencies to be meet and the code generator to be run. SPEST however strives to be as human readable as possible, and combats the problem of code littering by not requiring the kinds of annotations that JML does. In the same vein of human readability, SPEST generates substantially fewer tests than JML does, while retaining as much of the effectiveness of the tests as possible. Through the use of methods taken from various research journals, the effectiveness of tests is preserved while reducing the number of tests generated. As a result, the test generation can be completed in a much shorter time period while remaining nearly as effective a tool as the exhaustive tests generated by JML.

When testing software, there are often many cases that a tester needs to consider. Depending on the number of parameters a method accepts, the number of possible permutations that need to be tested increases quickly. SPEST provides a method for simplifying the testing of many of these permutations by generating the tests in software based on post and preconditions provided by the programmer. Tools like JML generate tests nearly exhaustively, generating tests in the millions. The SPEST language does its best to generate tests that are as efficient as possible. The goal of SPEST is to generate meaningful tests that are both human readable as well as effective, while not while not performing the more exhaustive test generation as is done with JML test generation tools..

1.2. Overview of the Solution

SPEST uses annotations embedded in Java code in order to generate its tests. SPEST

can generate all types of parameters based on these annotations, from primitives to arrays to objects. For each type of object generated, a generation technique is used in order to generate data that is as representative as possible. Users can specify a range of values for each field of their object or array, or specify no ranges at all and let the SPEST program decide how to generate the values to use in the parameter generation.

Once SPEST has generated it's test values, it can then generate readable unit tests that can run the method with the generated values, testing that the specified pre and post conditions are met for that method call. This allows classes to be tested easily simply by providing pre and post conditions for all class methods, and allowing SPEST to determine the values used to test. The creation of the Java unit tests is handled by SPEST as well, making testing a simpler task than tests developed manually by program developers. SPEST has advantages over similar tools such as JML because SPEST generates fewer tests, as well as tests that are more human readable. Tests can easily be modified since their structure is designed to be easy to understand.

1.3. Project Scope and Limitations

One of the main features that has yet reach full implementation is object generation. Object generation currently can generate objects to test with, but these objects are not guaranteed to meet the specified preconditions. More development of this system is needed in order to implement this important feature. In addition, a system for improving the variance of objects during testing is planned so that the range of tested values is sufficiently varied.

1.4. Outline of the Report

The remainder of the report contains information about usage of the SPEST test generator, and high level information about the design of SPEST. In addition, there is a step by step walk through of a usage scenario of spest using tutorial examples of an apple tree and a classroom. Finally, details of some of the unit testing used to test SPEST methods is discussed, as well as a discussion about related software and challenges that SPEST may face in the future.

2. Scenario of System Use

Two examples created in order to provide a tutorial for new users of SPEST are schoolroom and apple tree examples (code provided in Appendix A and Appendix B). These sample classes are simple, and provide a good starting point for someone who wants to understand how to get started with the SPEST programs.

The school example contains three classes: Student.java, Instructor.java, Course.java,

Implementer.java, and Room.java. Each of these classes has a few methods and fields that represent their respective objects (see the Java source code in Appendix A). This example has pre and post conditions already written into it, specifically in the Course.java class that demonstrate how one can specify pre and post conditions in order to specify what range of values should be generated by SPESTs black box test generation. The next step is to run the SPEST executable jar to bring up the SPEST GUI, as seen in Figure 1.

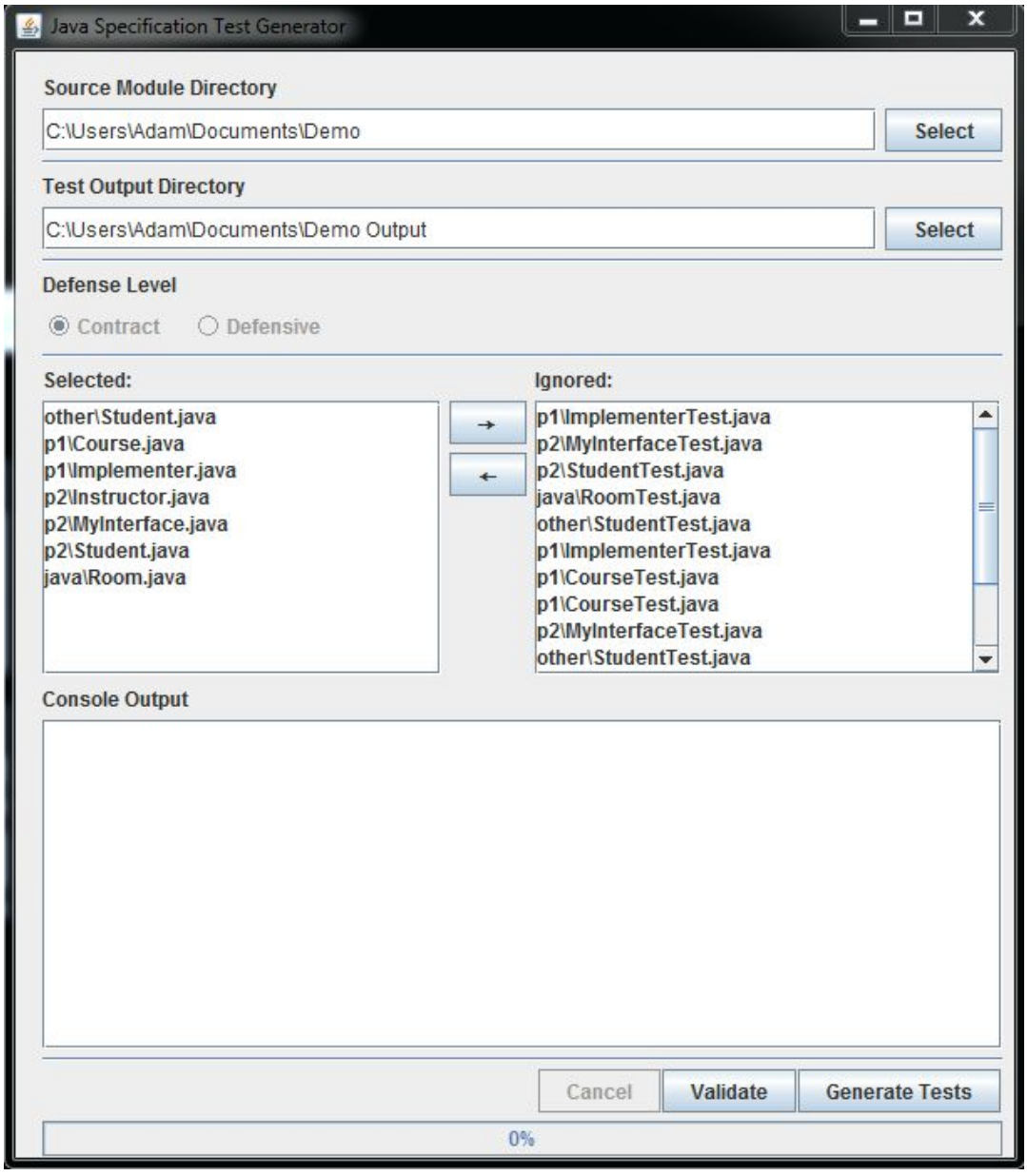

Figure 1. Using the SPEST program in the School Room example

The GUI will automatically fill with information about how to complete the run of the demo. There are a few important parts to take note of in the GUI, the “Selected” and “Ignored” boxes, as well as the “Source Module Directory” and “Test Output Directory”. The “Source

Module Directory will need to contain the absolute file path of the directory containing the classes that tests are to be generated for. The “Test Output Directory” will contain the absolute path to a directory where tests are to be placed after they are generated.

Classes in the “Selected” box are the ones that will have tests generated for them, and for this example it includes all of the .java classes in the demo. The final step is to run the test generation by clicking the “Generate Tests” button at the bottom of the GUI. Clicking this button will cause a progress bar at the bottom of the GUI to fill as classes are first validated, and then tests are generated for them.

```
@Test
public void sortStudentsTest_1() throws Exception
{
    java.util.List<other.Student> students = (java.util.List)cloner.deepClone(getFieldValue(testObj, "students"));


    Class[] parameterClasses = {};
    List<java.lang.Integer> indexs = javaTestUtility.getUniversalValues("sortStudents", parameterClasses, 0, testObj);


    testObj.sortStudents();


    for(int index : indexs)
    {
        Assert.assertTrue(testObj.students.get(index).polyId < testObj.students.get(index + 1).polyId);
    }
    setUp();
}
```

Figure 2. A test generated by SPEST for the School Room example

Figure 2 shows an example of a test generated by SPEST for a method called sortStudent. This test was generated by SPEST based on the pre and post conditions specified in the comments above the method signature for sortStudent in the source of Course.java. This test gets a list of students from the test object, runs sortStudent on the list, and then ensures that the post condition is true. For this test, the post condition requires that after the sortStudent method is called, each student in the list of students has an ID with a smaller value than the student in the following position in the list.

New tests will be generated in and placed into the specified test output directory. These tests will then be runnable JUnit tests, and can be executed like any other Java classes after compilation. For this specific demo example, tests for Course.java are generated. They include good examples of simple pre and post conditions, that in this specific case limit the the number of students that can be added to a class and what their IDs can be, ensure that the sort function of the Course class is functional, and even includes a failing test that makes a comparison by reference where an object equality tests is needed.

The Apple tree example is quite similar, but a bit simpler. It has only three classes, AppleTree.java which contains Apple objects, Apple.java, which contains Seed objects, and Seed.java which contain the primitive attributes of a seed (see the Java source code in Appendix B). The tests can be run in the same way that the previous school room example is

run, by executing the SPEST program, selecting input and output directories as well as .java files to include, and then clicking generate tests.

Tests generated for the Seed in the Apple Tree example are simpler than in the school room example, because most of the preconditions involve making sure that a new parameter is updated to a new value. The AppleTree.java class has tests that ensure that a null value aplpe cannot be added to an AppleTree, and that ensure that when a new apple is added, it is indeed added to the list of apples in the tree.

These two examples provide a simple demonstration of how to use SPEST, and the kinds of tests that SPEST can produce.

3. Spest System Design

SPEST's system design is divided into three main components Validation, Generation, and Execution, illustrated below in Figure 3.

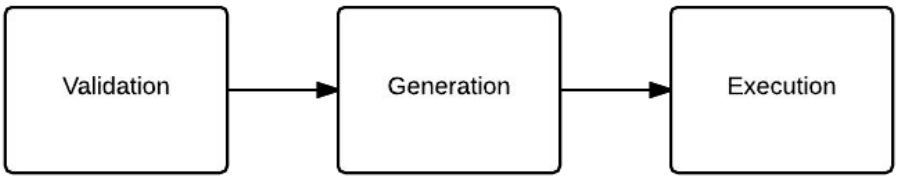

Figure 3. The three main stages of functionality in SPEST

The execution of SPEST consists of three main stages. These three phases are validation, test generation, and test execution. Two of these stages can be seen as buttons on the SPEST GUI in the Validate and Generate Tests button. Validation is the first stage of generating SPEST tests. Pre and post condition comments need to be checked for syntax and type information in order to make sure that the tests can be successfully generated. Once validity is determined, the next step is to generate the test classes. Test comments need to be parsed, and SPEST has methods that handle the conversion of these comments into executable java tests. Finally, the test can be compiled and run by a user. SPEST has a custom test runner that helps facilitate the generation of parameters that will be used in testing. Primitive and Object generation occur at this stage in the usage of SPEST.

4. Details of Primitive Value Generation

In order to be able to generate the full range of values that may be required by the pre and post conditions, it is necessary to create modules that can generate the required primitive

values given these conditions. For each parameter, an array of up to five values are generated. Ideally, these 5 values are as representative as possible of the range specified by the precondition. In order to achieve this goal, SPEST creates an array that contains the lowest value within the bound, one above the lowest value in the bound, the value at the center of the bound, the value at the top of the bound, and the value one off of the top of the bound. Each of the primitive data types of char, double, int, and boolean used this methodology when generating primitive values given a set of bounds. Since there are only two possible boolean values, the generation is simpler since only two unique values were a possibility for each parameter.

In addition to the traditional primitive values, array test values are also generated. For this phase of the generation, a similar method of generation is followed. SPEST generates five arrays of five values each, and applies the generation method between arrays as well as within arrays, where applicable. This means that SPEST generates low, mid, and high value arrays, and within each array varies all values from low, mid, to high. Again, the boolean value arrays are simpler compared to the double, char, and int arrays.

5. Unit Testing Spest Methods

Units of functionality that I was responsible for testing were the primitive generation modules for doubles, integers, double arrays and integer arrays, and the module that provides a simplified API for working with Java reflection.

For testing the primitive generation methods, the goal was to make sure that the methods generated the correct representative values for the given preconditions. Preconditions were fed to the methods, and then the primitives were generated. The resulting arrays were checked to make sure that they generated the correct values for the given preconditions, and that all of the values that they generated abided.

Testing of the simplified Java API in ReflexiveSupport.java mostly consisted of making sure that all of the methods in the class were robust, and could handle all kinds of bad inputs to their methods. The way that was agreed upon for handling bad inputs to these methods was to return null in all cases where any of the inputs were invalid. In addition, the tests needed to make sure that the simplified API could reflexively retrieve values from the provided classes in all of the same cases that the Java reflexive API does.

6. Related Work

JMLUnitNG is a test generation tool similar to SPEST that was created to improve upon JMLUnit [4]. Whereas JMLUnit was only able to generate primitive data for its test, JMLUnitNG has the capability to generate objects for its tests [4]. One of the key differences between JMLUnitNG and SPEST is the time taken to generate tests. JMLUnitNG can take hours to complete the test generation phase of its execution, where SPEST attempts to generate a few effective tests instead of trying to generate exhaustive tests. SPEST is able to provide feedback much more quickly to developers than JMLUnitNG is because of these differences. Unfortunately, JMLUnitNG is no longer being supported, and does not work with the most recent version of Java (Java 8). This exemplifies the need for newer tools like SPEST.

7. Conclusions and Future Work

My major contributions to the SPEST tool were the primitive generation and Unit Testing of a few of the SPESTS utility classes. The primitive generation component of SPEST is a very important feature of the tool. It allows SPEST to parse the pre and post conditions that a user specifies their value ranges with, and create values that are representative of the specified range. In addition, when object generation is added in the future, the object generation code will need to utilize primitive generation in order to function correctly, so it is quite an important component of the project.

My other main contribution was unit testing of utility classes and unit testing of primitive generation. The utility classes that I focused my testing on were the primitive generation classes, the classes that provided a simpler API for reflection, and a class that provided utility methods for the test generation feature.

One of the biggest features that SPEST does not have implemented yet is effective object generation. Currently, object generation does not always respect preconditions that are specified for those objects. In order to implement this new feature, objects will need to be generated and then check to see if they meet the precondition requirements. In order to get a sufficient number of objects to test, this process will have to be repeated until an adequate number of objects that meet the preconditions have been filtered out. Generation of objects in a quick manner is one of the key features that sets SPEST apart from its competitors, so this feature is a rather important one.

Appendix A: School Room Example Source

Instructor.java:

```
package p1.p2;

/**
 * Created by cjohnson on 4/28/15.
 */
public class Instructor
{
   private String name;

   public Instructor()
   {
   }

   public String getName()
   {
      return name;
   }
```

```
    public void setName(String name)
    {
        this.name = name;
    }

    @Override
    public boolean equals(Object o)
    {
        if (this == o)
        {
            return true;
        }
        if (o == null || getClass() != o.getClass())
        {
            return false;
        }

        Instructor that = (Instructor) o;

        return !(name != null ? !name.equals(that.name) : that.name != null);

    }

    @Override
    public int hashCode()
    {
        return name != null ? name.hashCode() : 0;
    }
}
```

Course.java:

```
package p1;

import other.Student;
import p1.p2.Instructor;
import support.reflection.ReflexiveSupport;

import java.io.File;
import java.io.IOException;
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import java.util.*;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
```

```
 * Created by cjohnson on 4/27/15.
 */
public class Course
{
    public static final int MAX = 5;
    public HashMap<String, List<Course>> courseHashMap;
    public List<Student> students;

    public int courseNumber;

    private Instructor instructor;

    public static void main(String[] args)
    {
        HashMap hashMap = new HashMap();

        System.out.println(Arrays.toString(HashMap.class.getTypeParameters()));

        Type t1 = HashMap.class.getTypeParameters()[1];
        Type t2 = HashMap.class.getTypeParameters()[1];

        System.out.println(t1.equals(t2));
    }

    public Course(Instructor instructor, int courseNumber)
    {
        courseHashMap = new HashMap<>();

        this.courseNumber = courseNumber;
        this.instructor = instructor;

        students = new ArrayList<Student>();
    }

    /*
    PRE:
        student != null
        && student.polyId > 100
    POST:
        forall(int index; index >= 0 && index < students.size();
            students.contains(students'.get(index)) || students'.get(index).equals(student)
        )
        && students.size() == students'.size() - 1
        && students.size() <= MAX
        && Integer.MAX_VALUE == Integer.MAX_VALUE
     */
    public void addStudent(Student student)
    {
```

```
        students.add(student);
    }

    /*
    POST:
        forall(int index; index >= 0 && index < students.size() - 1;
            students'.get(index).polyId < students'.get(index + 1).polyId
        )
     */
    public void sortStudents()
    {
        Comparator<Student> comparator = new Comparator<Student>()
        {
            @Override
            public int compare(Student o1, Student o2)
            {
                return Integer.compare(o1.getPolyId(), o2.getPolyId());
//                return o1.getPolyId() - o2.getPolyId();
            }
        };

        Collections.sort(students, comparator);
    }

    /*
    POST:
        return.equals(instructor)
     */
    public Instructor getInstructor()
    {
        return instructor;
    }

    /*
    POST:
        //This will fail since it is a reference comparison
        return == students
        && ReflexiveSupport.isGeneric(null);
     */
    public List<Student> getStudents()
    {
        return students;
    }

    /*
    POST:
        return == courseNumber
     */
```

```
    public int getCourseNumber()
    {
        return courseNumber;
    }


    /*
    POST:
        this.instructor'.equals(instructor)
     */
    public void setInstructor(Instructor instructor)
    {
        this.instructor = instructor;
    }

    /*
    POST:
        (courseNumber == courseNumber')
        && if(!(courseNumber == courseNumber'))
            (string != null)
        else
            ((string == null))
        &&
        (courseNumber != courseNumber')
     */
    public void testIfStatement(int courseNumber, String string)
    {

    }

    /*
    POST:
        this.courseHashMap' == courseHashMap
        && courseHashMap.get("TEST").get(0).courseNumber == example.courseNumber;
     */
    public void testHashMap(HashMap<String, List<Course>> courseHashMap, Course
example)
    {
//        this.courseHashMap = courseHashMap;
    }

    /*
    POST:
        true
        && !(MAX != MAX)
        && (((courseNumber == courseNumber)
        && !(courseNumber != courseNumber))
        || (courseNumber != courseNumber))
```

```
     */
    public void testParentheses()
    {

    }
}
```

Room.java:

```
/**
 * Created by cjohnson on 4/28/15.
 */
public class Room
{
    private int buildingNum;
    private int roomNum;

    public Room(int buildingNum, int roomNum)
    {
        this.buildingNum = buildingNum;
        this.roomNum = roomNum;
    }

    public int getBuildingNum()
    {
        return buildingNum;
    }

    public void setBuildingNum(int buildingNum)
    {
        this.buildingNum = buildingNum;
    }

    public int getRoomNum()
    {
        return roomNum;
    }

    public void setRoomNum(int roomNum)
    {
        this.roomNum = roomNum;
    }
}
```

Appendix B: Apple Tree Example Source

Apple.java:

```
package sampleCode.FisherSample;

import java.awt.*;
import java.util.List;

public class Apple
{
   private Color color;
   private List<Seed> seeds;
   private double weight;

   public Apple(Color color, List<Seed> seeds, double weight)
   {
      this.color = color;
      this.seeds = seeds;
      this.weight = weight;
   }

   /*
   PRE:
      weight > 1.5
   POST:
      weight' == (weight - 1.5)
    */
   public void takeBite()
   {
      weight -= 1.5;
   }

}
```

AppleTree.java:

```
package sampleCode.FisherSample;

import java.util.List;

public class AppleTree
{
   private List<Apple> apples;

   public AppleTree(List<Apple> apples)
   {
      this.apples = apples;
   }
```

```
    /*
    PRE:
        apple != null
    POST:
        forall(Apple iterateApple; apples.contains(iterateApple);
                apples'.contains(iterateApple) || iterateApple.equals(apple)
              )
     */
    public void addApple(Apple apple)
    {
        apples.add(apple);
    }

    /*
    PRE:
        apple != null
    POST:
        forall(Apple iterateApple; apples.contains(iterateApple);
                apples'.contains(iterateApple) && !iterateApple.equals(apple)
              )
     */
    public void removeApple(Apple apple)
    {
        apples.remove(apple);
    }
}
```

Seed.java:

```
package sampleCode.FisherSample;

import java.awt.*;

public class Seed
{
    private Color color;
    private double length;
    private int weight;
    private boolean isRound;

    public Seed(Color color, double length, int weight, boolean isRound)
    {
        this.color = color;
        this.length = length;
        this.weight = weight;
        this.isRound = isRound;
    }
```

```
/*
POST:
    return == color
 */
public Color getColor()
{
    return color;
}

/*
POST:
    color' == color
 */
public void setColor(Color color)
{
    this.color = color;
}

/*
POST:
   return == color
*/
public double getLength()
{
    return length;
}

/*
PRE:
    length < 35.4 && length > 25.6
    || length <= 45.1 && length >= 35.9
POST:
    length' == length
*/
public void setLength(double length)
{
    this.length = length;
}

/*
POST:
   return == weight
*/
public int getWeight()
{
    return weight;
}
```

```
    /*
    POST:
       length' == length
    */
    public void setWeight(int weight)
    {
       this.weight = weight;
    }

    /*
    POST:
       return == isRound
    */
    public boolean isRound()
    {
       return isRound;
    }

    /*
    POST:
       isRound' == isRound
    */
    public void setRound(boolean isRound)
    {
       this.isRound = isRound;
    }
}
```

References

1. Association for Computing Machinery, Computing Surveys, 1995-2014.

2. Association for Computing Machinery, International Symposium on Software Testing and Analysis, 1997-2013,

3. IEEE Computer Society, Transactions on Software Engineering, 1992-2005.

4. Leavens, Gary T. et. all, JMLUNIT, http://www.eecs.ucf.edu/~leavens/JML-release/docs/man/jmlunit.html

5. Leavens, Gary T. et al., JML Reference Manual, eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html, May 2013

6. Zimmerman, Daniel, "JMLUnit: The Next Generation", Formal Verification of Object-Oriented Software, Lecture Notes in Computer Science, Volume 6528, pp 183-197, 2011.